A taxonomy of task-based parallel programming technologies for high-performance computing

J Supercomput
https://doi.org/10.1007/s11227-018-2238-4

A taxonomy of task-based parallel programming technologies for high-performance computing

Peter Thoman[1] · Kiril Dichev[2] · Thomas Heller[3] · Roman Iakymchuk[4] · Xavier Aguilar[4] · Khalid Hasanov[5] · Philipp Gschwandtner[1] · Pierre Lemarinier[5] · Stefano Markidis[4] · Herbert Jordan[1] · Thomas Fahringer[1] · Kostas Katrinis[5] · Erwin Laure[4] · Dimitrios S. Nikolopoulos[2]

Abstract Task-based programming models for shared memory—such as Cilk Plus and OpenMP 3—are well established and documented. However, with the increase in parallel, many-core, and heterogeneous systems, a number of research-driven projects have developed more diversified task-based support, employing various programming and runtime features. Unfortunately, despite the fact that dozens of different task-based systems exist today and are actively used for parallel and high-performance computing (HPC), no comprehensive overview or classification of task-based technologies for HPC exists. In this paper, we provide an initial task-focused taxonomy for HPC technologies, which covers both programming interfaces and runtime mechanisms. We

This work was supported by the AllScale EC-funded FET-HPC H2020 Project (No. 671603).

✉ Roman Iakymchuk
riakymch@kth.se

Peter Thoman
petert@dps.uibk.ac.at

Kiril Dichev
K.Dichev@qub.ac.uk

Thomas Heller
thomas.heller@fau.de

Xavier Aguilar
xaguilar@kth.se

Khalid Hasanov
khasanov@ie.ibm.com

Philipp Gschwandtner
philipp@dps.uibk.ac.at

Pierre Lemarinier
pierrele@ie.ibm.com

Published online: 12 January 2018

demonstrate the usefulness of our taxonomy by classifying state-of-the-art task-based environments in use today.

Keywords High-performance computing · Task-based parallelism · Taxonomy · API · Runtime system · Scheduler · Monitoring framework · Fault tolerance

1 Introduction

A large number of task-based programming environments have been developed over the past decades, and the task-based parallelism paradigm has proven widely applicable for consumer applications. Conversely, in high-performance computing (HPC), loop-based and message-passing paradigms are still dominant. In this work, we specifically aim to categorize task-based parallelism technologies which are in use in HPC.

For the purpose of this work, we define a *task* as follows

> A **task** is a sequence of instructions within a program that can be processed concurrently with other tasks in the same program. The interleaved execution of tasks may be constrained by control- and data-flow dependencies.

Many programming languages support task-based parallelism directly without external dependencies. Examples include the C++11 thread support library, Java via its concurrency API, or Microsoft TPL for .NET. Except for C++, we do not study these languages in detail in this paper, since they are not common in the HPC domain.

Stefano Markidis
markidis@kth.se

Herbert Jordan
herbert@dps.uibk.ac.at

Thomas Fahringer
tf@dps.uibk.ac.at

Kostas Katrinis
katrinisk@ie.ibm.com

Erwin Laure
erwinl@kth.se

Dimitrios S. Nikolopoulos
D.Nikolopoulos@qub.ac.uk

1 University of Innsbruck, 6020 Innsbruck, Austria

2 Queen's University of Belfast, Belfast BT7 1NN, UK

3 University of Erlangen-Nürnberg, 91058 Erlangen, Germany

4 KTH Royal Institute of Technology, 100 44 Stockholm, Sweden

5 IBM Ireland, Dublin 15, Ireland

The Cilk language[1] [5] allows task-focused parallel programming and is an early example of efficient task scheduling via work stealing. OpenMP [10], which we consider a language extension, integrates tasks into its programming interface since version 3.0. In addition to *languages* and *extensions*, industry-standard and well-supported parallel *libraries* based on task parallelism have emerged, such as Intel Cilk Plus [23] or Intel TBB [28]. There are also *runtimes* specifically designed to improve shared memory performance of existing language extensions, such as Qthreads [27] or Argobots [24]; this topic is of significant importance, considering the increase in many-core processors in recent years and, consequently, the importance of efficient lightweight runtimes. Task-based environments for heterogeneous hardware have also naturally developed with the emergence of accelerator and GPU computing; StarPU [2] is an example of such an environment.

In addition, task-based parallelism is increasingly employed on distributed memory systems, which constitute the most important target for HPC. In this context, tasks are often combined with a global address space (GAS) programming model such as GASPI [25] and scheduled across multiple processes, which together form the *distributed execution* of a single task-parallel program. While some examples of global address space environments with task-based parallelism are specifically designed languages such as Chapel [7] and X10 [8], it is also possible to implement these concepts as a library. For instance, HPX [17] and Charm++ [18] are asynchronous GAS runtimes.

This already very diverse landscape is made even more complex by the recent appearance of task-based runtimes using novel concepts, such as the data-centric programming language Legion [3]. Many of these task-based programming environments are maintained by a dedicated community of developers and are often research oriented. As such, there might be relatively little accessible documentation of their features and inner workings.

Crucially, at this point, *there is no up-to-date and comprehensive taxonomy and classification of existing common task-based environments*. Consequently, researchers and developers with an interest in task-based HPC software development cannot obtain a concise picture of the alternatives to the omnipresent MPI programming model. In this work, we attempt to address this issue by first providing a taxonomy of task-based parallel programming environments. The applicability of this taxonomy is then validated by applying it to classify a number of task-based programming environments. While not all of these environments are equally mature and stable, they build an important snapshot of the task-based APIs and runtimes in use in HPC today. We consider that each task-based environment has two central components: a *programming interface* (API) and a *runtime system*; the former is the interface that a given environment provides to the programmer, while the latter encompasses the underlying implementation mechanisms.

The remainder of this article is organized as follows. In Sect. 2, we present a set of API characteristics allowing meaningful classification. For discussing the more involved topic of runtime mechanisms, in Sect. 3, we further structure our analysis into the overarching topics of scheduling (Sect. 3.1), performance monitoring (Sect. 3.2),

[1] Note that we use the term "language" for Cilk and Cilk Plus, even though they build on C/C++. The reasoning is that a Cilk Plus compiler is strictly required for compilation (unlike, for example, OpenMP).

and fault handling (Sect. 3.3). Based on the taxonomy introduced, we classify and categorize existing APIs and runtimes in Sect. 4. Finally, Sect. 5 concludes the article.

2 Task-parallel programming interfaces

The application programming interface (API) of a given task-parallel programming environment defines the way an application developer describes parallelism, dependencies, and in many cases other more specific information such as the work mapping structure or data distribution options. As such, finding a way to concisely characterize APIs from a developer's perspective is crucial in providing an overview of task-parallel technologies.

In this work, we define a set of characterizing features for such APIs which encompasses all relevant aspects while remaining as compact as possible; Fig. 1 presents our API taxonomy. A subset of these features was adapted from previous work by Kasim et al. [19]. To these existing characteristics, we added additional information of general importance—such as technological readiness levels—as well as features which relate to new capabilities particularly relevant for modern HPC like support for heterogeneity and resilience management.

We have grouped these aspects in four broad categories: those describing the **architectural** aspects targeted by an environment; those that summarize the **task system**

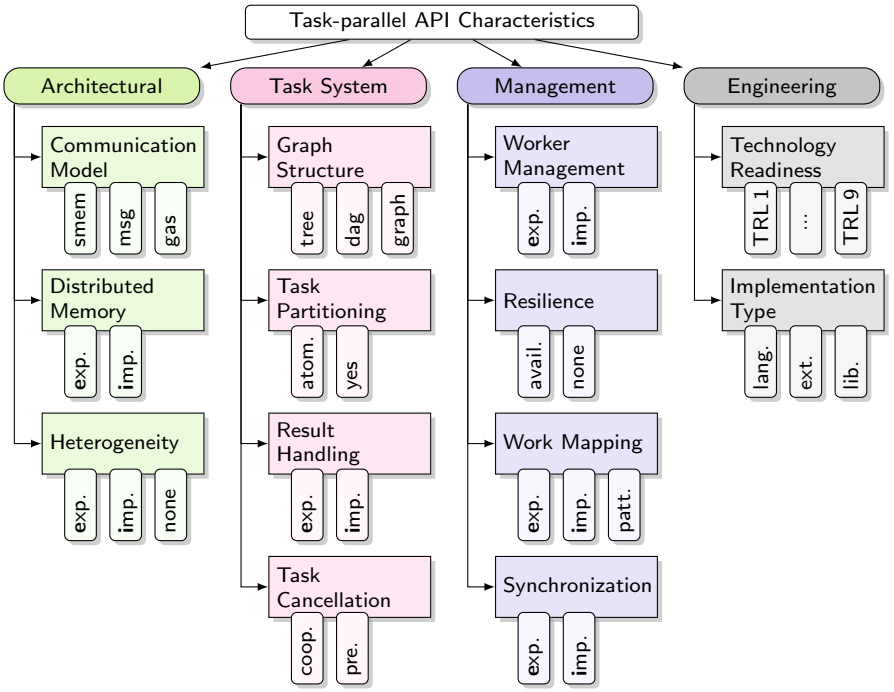

Fig. 1 Taxonomy of APIs

offered by an API; aspects related to the **management** of work; and finally **engineering** information. For many of these characteristics, *explicit* (**e**) support refers to features which require extra effort or implementation by the developer, while *implicit* (**i**) support means that the toolchain manages the feature automatically.

We list below these four categories with the corresponding aspects.

2.1 Architectural

Communication model	Either *shared memory* (smem), *message passing* (msg), or a *global address space* (gas).
Distributed memory	Whether targeting distributed memory systems is supported. Options are *no* support, *explicit* support, or *implicit* support. *explicit* refers to, for example, message passing between address spaces, while automatic data migration would be an example of *implicit* support.
Heterogeneity	This indicates whether tasks can be executed on accelerators (e.g. GPUs). *Explicit* support indicates that the application developer has to actively provision tasks to run on accelerators, using a distinct API.

2.2 Task system

Graph structure	The type of task graph dependency structure supported by the given API. Possibilities include a *tree* structure, an *acyclic graph* (dag), or an *arbitrary graph*.
Task partitioning	This feature indicates whether each task is atomic—can, thus, only be scheduled as a single unit—or can be subdivided/split.
Result handling	Whether the tasking model features *explicit* handling of the results of task computations. For example, return types accessed as *futures*.
Task cancellation	Whether the tasking model supports cancellation of tasks. Options are *no* cancellation support; cancellation is supported either *cooperatively* (only at task scheduling points) or *preemptively*.

2.3 Management

Worker management	Whether the worker threads and/or processes need to be started and maintained by the user (*explicit*) or are provided automatically by the environment (*implicit*).
Resilience management	This feature describes whether the API has support for task resilience management, e.g. fine-grained checkpointing and restart.

Work mapping	This feature describes the way tasks are mapped to the existing hardware resources. Possibilities include *explicit* work mapping, *implicit* work mapping (e.g. stealing), or *pattern-based* work mapping.
Synchronization	Whether tasks are synchronized in an *implicit* fashion, e.g. by regions or the function scope, or *explicitly* by the application developer.

2.4 Engineering

Technology readiness	The technology readiness of the given API according to the European Commission definition.[2] If an API has multiple implementations, the most mature one is used to assess this metric.
Implementation type	How the API is implemented and addressed from a program. A tasking API can be provided either as a *library*, a *language extension*, e.g. pragmas, or an entire *language* with task integration.

3 Many-task runtime systems

Many-task runtime systems serve as the basis for implementing the APIs described in Sect. 2 and are considered a promising tool in addressing key issues associated with Exascale computing. In this section, we provide a taxonomy of many-task runtime systems, which is illustrated in Fig. 2.

A crucial difference among various many-task runtime systems is their **target architecture**. The evolution of many-task runtime systems started from *homogeneous shared memory* computers with multiple cores and continued towards runtimes for *heterogeneous* shared memory and *distributed memory* platforms. Support for distributed memory architectures varies significantly across different systems: in case of *implicit data distribution*, data distribution is handled by the runtime, without putting any burden on the application developer; on the other hand, *explicit data distribution* means that distribution across nodes is explicitly specified by the programmer.

Modern HPC systems require efficiency not only in execution times, but also in power and/or energy consumption. Thus, whether the runtime provides **scheduling targets** (Sect. 3.1.1) other than the execution time is another important distinction between different runtimes. At the same time, the runtime can achieve its scheduling target by using different **scheduling methods** (Sect. 3.1.2). We divide them into three categories, namely *static*, *dynamic*, and *hybrid* scheduling methods. Some of them provide automatic scheduling within a single shared memory machine, while the application developer needs to handle distributed memory execution explicitly, whereas others provide uniform scheduling policies across different nodes.

[2] https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf. Accessed: 2017-12-16.

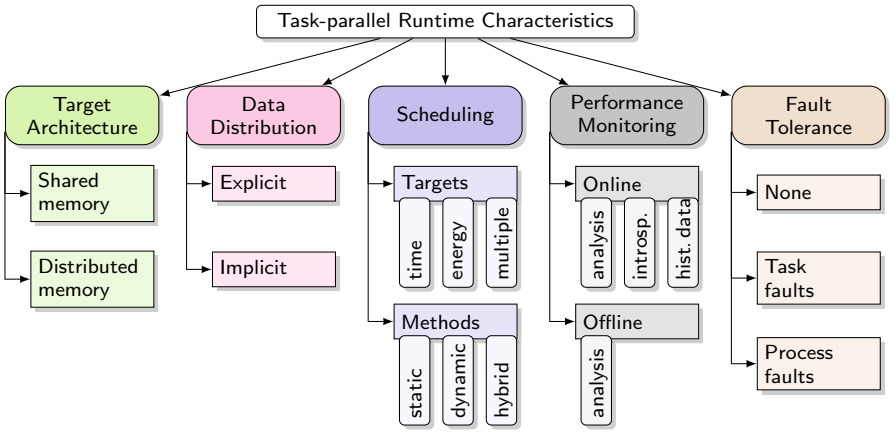

Fig. 2 Taxonomy of many-task runtime systems

Many-task runtimes may require **performance introspection** and **monitoring** (Sect. 3.2) to facilitate the implementation of different scheduling policies, that is, using online performance information to assist the decision making process of the scheduler. **Fault tolerance** is another key factor that is important in many-task runtime systems in the context of Exascale requirements. As detailed in Sect. 3.3, a runtime may have no resilience capabilities, or it may target task faults or even process faults.

3.1 Scheduling in many-task runtime systems

3.1.1 Scheduling targets

Depending on the capabilities of the underlying many-task runtime system, its scheduling domain is usually limited to a single shared memory homogeneous compute node, a heterogeneous compute node with accelerators, homogeneous distributed memory systems of interconnected compute nodes, or in a most generic form to heterogeneous distributed memory systems. By supporting different types of heterogeneous architectures, the runtime can facilitate source code portability and support transparent interaction between different types of computation units for application developers.

Traditionally, *execution time* has been the main objective to minimize for different scheduling policies. However, the increasing scale of HPC systems makes it necessary to take the energy and power budgeting of the target system into account as well. Therefore, some many-task runtime systems have already started providing *energy-aware* [20] scheduling policies.[3] In a simple scenario, it is assumed that the application can provide an energy consumption model which can be used by a scheduling policy as part of its objective function. In more advanced cases, the runtime provides offline or online profiling data. These data are used to build a look-up table that maps each

[3] http://starpu.gforge.inria.fr/doc/html/Scheduling.html#Energy-basedScheduling.

frequency setting with profiling data and the number of active cores. Then, a scheduling decision is made based on this information.

In addition, recent research projects such as AllScale[4] focus on *multi-objective* scheduling policies trying to find promising trade-off solutions among conflicting optimization objectives like execution time, energy consumption, and/or resource utilization.

3.1.2 Scheduling methods

Extensive research has been conducted in task scheduling methodologies. We simply highlight the state-of-the-art scheduling methods used in many-task runtime systems, without providing an exhaustive overview. The distribution of work at compile time is *static scheduling*, while the distribution of work at runtime is *dynamic scheduling*.

For static scheduling, depending on the decision function, either one or more of the following inputs are known in advance: execution time of each task, inter-dependencies between tasks, task precedence, resource usage of each task, location of the input data, task communications, and synchronization points. Static scheduling algorithms used in many-task runtime systems are based on distributing a global task list to different compute units; a typical example is a round-robin distribution.

On the other hand, dynamic scheduling is used if the requested information is not available before execution, or obtaining such information is too complex. Possibly, the most important choice for dynamic scheduling is load balancing; its use is a complex trade-off between its cost and its benefits. Work stealing [4] can be considered the most widely used load balancing technique in task-based runtime systems. The main concept in work stealing is to distribute tasks in per-processor work queues, where each processor operates on its local queue. The processors can steal tasks from other queues to perform load balancing. Another approach to dynamic scheduling for many-task runtime systems is the work sharing strategy. Unlike work stealing, it schedules each task onto a processor when it is spawned and it is usually implemented by using a centralized task pool. In work sharing, whenever a worker spawns a new task, the scheduler migrates it to a new worker to improve load balancing. As such, migration of tasks happens more often in work sharing than in work stealing.

Combinations between static and dynamic scheduling are possible. Static scheduling can be adapted and used for dynamic scheduling by re-computing and re-sequencing the list of tasks. Indeed, heuristic policies based on list scheduling and performance models are employed in some many-task runtime systems [2]. Additionally, *hybrid* policies, which integrate static and dynamic information, are possible.

3.2 Performance monitoring

The high concurrency and dynamic behaviour of upcoming Exascale systems poses a demand for performance observation and runtime introspection. This performance information is very valuable to dynamically steer many-task runtime systems in their

[4] The AllScale EC-funded FET-HPC project: allscale.eu.

execution and resource adaption, thereby improving application performance, resource utilization, or energy consumption.

When targeting performance observation, performance monitoring software is either generating data to be used *online* [1,3,14,15,22] or *offline* [2,3,11,14]. In other words, whether the collected data are going to be used, while the application still runs or after its execution. Furthermore, this taxonomy can be extended with respect to who is consuming data—either the end user (*performance analysis*) or the runtime itself (*introspection* and *historical data*). Real-time performance data (*introspection* and performance models from *historical data*) will play an important role in Exascale for runtime adaptation and optimal task scheduling.

3.3 Fault tolerance

In order to detail what level of fault tolerance a runtime may have, we need to identify what types of faults we anticipate. For this topic, we *extend* a taxonomy [13] from the HPC domain to include the concept of task faults. We retain detectability of faults as the main criterion, but distinguish three levels of the system: distributed execution, process, and task. Each of these levels may experience a fault, and each of them has a different scope. Only task faults and process faults can possibly be detected from within an application. Moreover, only for these types of faults some recovery mechanisms can be implemented at the task or process level inside a runtime system.

3.3.1 Task faults

Tasks have the smallest scope of the three; still, a failure of a task may affect the result of a process and subsequently of a distributed run. A typical example is undetected errors in memory. The process which runs a task is generally capable of detecting task faults. There are several examples of shared memory runtimes which are capable of detecting and correcting task faults within parallel regions [16,26].

3.3.2 Process faults

A process may also fail, which leads to the termination of all underlying tasks. For example, a node crash can lead to a process failure. In such a scenario, a process cannot detect its failure; however, in a distributed run, another process may detect the failure and trigger a recovery strategy across all processes. A recovery strategy in this case may rely on one of two redundancy techniques: checkpoint/restart or replication.

3.3.3 System faults

On the last level, a distributed system execution may fail in cases of severe faults like switch failure or power outage. While failure detection such as power outage detection can be placed at the system level, most of them cannot be detected directly by the runtime. Recovery mechanisms can take the form of a global checkpoint restart of the entire application.

Table 1 Feature comparison of APIs for task parallelism

	Architectural			Task System				Management				Eng.	
	Communication Model	Distributed Memory	Heterogeneity	Graph Structure	Task Partitioning	Result Handling	Task Cancellation	Worker Management	Resilience Management	Work Mapping	Synchronization	Technological Readiness	Implementation Type
C++ STL	smem	×	×	dag	×	i/e	×	i	×	i/e	e	9	Library
TBB	smem	×	×	tree	×	i	✓	i	×	i	i	8	Library
HPX	gas	i	e	dag	✓	e	✓	i/e	×	i/e	e	6	Library
Legion	gas	i	e	tree	✓	e	×	i	×	i/e	e	4	Library
PaRSEC	msg	e	e	dag	×	e	✓	i	✓	i/e	i	4	Library
OpenMP	smem	×	i	dag	×	i	✓	e	×	i	i/e	9	Extension
Charm++	gas	i	e	dag	✓	i/e	×	i	✓	i/e	e	6	Extension
OmpSs	smem	×	i	dag	×	i	×	i	✓	i	i/e	5	Extension
AllScale	gas	i	i	dag	✓	i/e	×	i	✓	i	i/e	3	Extension
StarPU	msg	e	e	dag	✓	i	×	i	×	i/e	e	5	Extension
Cilk Plus	smem	×	×	tree	×	i	×	i	×	i	e	8	Lang.
Chapel	gas	i	i	dag	✓	i	×	i	×	i/e	e	5	Lang.
X10	gas	i	i	dag	✓	i	×	i	✓	i/e	e	5	Lang.

4 Classification

Following the API taxonomy defined in Sect. 2, Table 1 classifies existing task-parallel APIs. Note that for an API to qualify as supporting a given feature, this API must not require the user to resort to third-party libraries or implementation-specific details of the API. For instance, some APIs offer arbitrary task graphs via manual task reference counting [12]. This does not qualify as support in our classification. Also note that all APIs marked as featuring task cancellation do so in a non-preemptive manner due to the absence of OS-level preemption capabilities.

Some entries require additional clarification. In C++ STL, we consider the entity launched by `std::async` to represent a task. At the same time, HPX is an implementation of the C++ tasking API providing additional features such as distributed memory support and task dependencies. Also, while StarPU offers shared memory parallelism, it is capable of generating MPI communication from a given task graph and data distribution [2]; hence, it is marked with explicit support for distributed memory using a message-based communication model. Furthermore, PaRSEC includes both a task-based runtime that works on user-specified task graph and data distribution information, as well as a compiler that accepts serial input and generates this data. As the latter is limited to loops, we only consider the runtime in this work.

Several observations can be made from the data presented in Table 1. First, all APIs targeting distributed memory also support heterogeneity in some form. APIs offering implicit distributed memory support generally employ a global address space and implement task partitioning. Second, among APIs lacking distributed memory, only OmpSs offers resilience (via its Nanos++ runtime), and distributed memory APIs only recently started to include resilience support [9]—likely driven by the continuous increase in machine sizes and, hence, decreased mean time between failures. Finally,

Table 2 Feature comparison of runtimes for task parallelism

	Target Architecture	Data Distribution	Scheduling Methods (**sm**)	Scheduling Methods (**d**)	Performance Monitoring	Fault Tolerance
OpenMP runtimes(*)	**sm**	×	**m**	×	**off**/**on**	×
Intel TBB	**sm**	×	**ws**	×	**off**	×
Intel Cilk Plus	**sm**	×	**ws**	×	**off**	×
StarPU	**sm**	**e**	**m**	×	**off**/**on**	×
Nanos++	**d**	**i**	**m**	×	**off**/**on**	**tf**
Charm++	**d**	**i**	**m**	**m**	**off**/**on**	**pf**
X10	**d**	**i**	**ws**	×	**off**	**pf**
Chapel	**d**	**i**	**ws**	×	**off**	**pf**
HPX	**d**	**i**	**m**	×	**off**/**on**	×
AllScale	**d**	**i**	**ws**	**l**	**off**/**on**	**pf**
ParSEC	**d**	**e**	**m**	**l**	**off**	**tf**
Legion	**d**	**i**	**ws**	**ws**	**off**/**on**	**pf**

i implicit
e explicit
m multiple (incl. ws)
l limited
ws work stealing
tf task faults
pf process faults
sm shared memory
d distributed memory
on online use
off offline use

(*) Such as Intel OpenMP, GOMP, Qthreads, and Argobots

some form of heterogeneity support is provided in almost all modern APIs, though it often requires explicit heterogeneous task provisioning by the programmer.

Table 2 provides the corresponding classification with respect to the runtime system and its subcomponents taxonomy, introduced in Sect. 3. It is important to note that an API always translates into at least one runtime implementation, but is not always limited to one such implementation. The most diverse example is offered by the OpenMP API, which has many runtime implementations. We have grouped together most shared memory OpenMP implementations, due to them offering the same feature set in terms of our classification. However, Nanos++ is listed separately, since it goes beyond that, both in supported pragmas and in its support for distributed execution. Other runtimes with shared memory and distributed memory support (e.g. HPX) can also run as backends to OpenMP programs, but we do not detail these features in relation to OpenMP further. For the majority of cases, except for the OpenMP API and the Cilk API, there is a 1:1 mapping between API and runtime.

We refrain from using the scheduling targets in the runtime table and only include the scheduling methods. This decision is driven by the fact that energy has only recently become an important factor for scheduling, and energy-driven scheduling targets, or multi-objective scheduling targets, are only now starting to emerge. We list separately the scheduling methods for shared memory, and distributed memory, since these can be implemented in different ways. Indeed, most of the listed runtimes have similarities in scheduling within a single node; work stealing is the most common method of scheduling in this case. On the other hand, there is no established method for inter-node scheduling. For instance, ParSEC [6] only provides a limited inter-node scheduling based on remote completion notifications, while Legion uses distributed work stealing. AllScale is also designed to support distributed work stealing, but as its current implementation is incomplete it is marked as supporting limited inter-node scheduling.

Again, as for the APIs, we note that there are various contributions extending runtime features, but if these contributions are not part of the main release, they are not considered in our taxonomy. For instance, recent work in X10 [21] extends the X10 scheduler with distributed work stealing algorithms across nodes; however, we classify X10 as not (yet) having a distributed scheduler. The same applies to StarPU and OmpSs. New distributed memory scheduling policies are being developed for these runtimes, but they are not part of their main release yet.[5] Also, for Chapel, X10, and HPX, there is automatic data distribution support (runtime feature); however, these runtimes require explicit work mapping in distributed memory environments (API feature).

5 Conclusions

The shift in HPC towards task-based parallel programming paradigms has led to a broad ecosystem of different task-based technologies. With such diversity, and some degree of isolation between individual communities of developers, there is a lack of documentation and common classification, thus hindering researchers who wish to obtain a comprehensive view of the field. In this paper, we provide an initial attempt at establishing a common taxonomy and providing the corresponding categorization for many existing task-based programming environments suitable for HPC.

We divided our taxonomy into two broad categories: *APIs*, which define how the programmer interacts with the system, and *many-task runtime systems*, covering the underlying technologies and implementation mechanisms. For the former, we identify four broad categories of features exposed to the programmer: architectural characteristics; those related to the task generation and handling; system-level management; and finally engineering aspects. For the latter, we analyse the types of scheduling policies and goals supported; online and offline performance monitoring integration; as well as the level of resilience and detection provided for task, process, and system faults.

We believe that this paper provides a useful basis to describe task-based parallel programming technologies and to select, examine, and compare APIs and runtime systems with respect to their capabilities. This serves as a foundation to both classify additional APIs and runtime systems using our definitions as well as to allow for a better overview and comparative basis for newly implemented features within the ever-expanding, diverse field of task-based parallel programming environments.

References

1. Augonnet C et al (2009) Automatic calibration of performance models on heterogeneous multicore architectures. In: Proceedings of Euro-Par 2009, pp 56–65. Springer

[5] We received feedback from their developers.

2. Augonnet C et al (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concur Comput Pract Exp 23(2):187–198
3. Bauer ME (2014) Legion: programming distributed heterogeneous architectures with logical regions. Ph.D. thesis, Stanford University
4. Blumofe R, Leiserson C (1999) Scheduling multithreaded computations by work stealing. J ACM (JACM) 46(5):720–748
5. Blumofe RD et al (1996) Cilk: an efficient multithreaded runtime system. J Parallel Distrib Comput 37(1):55–69
6. Bosilca G et al (2013) Parsec: exploiting heterogeneity to enhance scalability. Comput Sci Eng 15(6):36–45. https://doi.org/10.1109/MCSE.2013.98
7. Chamberlain BL et al (2007) Parallel programmability and the chapel language. Int J HPC Appl 21(3):291–312
8. Charles P et al (2005) X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of OOPSLA05, pp 519–538. ACM. https://doi.org/10.1145/1094811.1094852
9. Cunningham D et al (2014) Resilient x10: efficient failure-aware programming. In: Proceedings of PPoPP14, pp 67–80. ACM. https://doi.org/10.1145/2555243.2555248
10. Dagum L, Menon R (1998) Openmp: an industry standard api for shared-memory programming. Comput Sci Eng 5(1):46–55
11. Duran A et al (2011) Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Process Lett 21(02):173–193
12. General Acyclic Graphs of Tasks in TBB. https://software.intel.com/en-us/node/506110. Accessed 16 Nov 2017
13. Hoemmen M, Heroux M (2011) Fault-tolerant iterative methods via selective reliability. In: Proceedings of SC11, p 9. ACM
14. Huck K et al (2013) An early prototype of an autonomic performance environment for exascale. In: Proceedings of ROSS13, p 8. ACM
15. Huck K et al (2015) An autonomic performance environment for exascale. Supercomput Front Innov 2(3):49–66
16. Hukerikar S et al (2014) Opportunistic application-level fault detection through adaptive redundant multithreading. In: Proceedings of HPCS14, pp 243–250. IEEE
17. Kaiser H et al (2014) Hpx: a task based programming model in a global address space. In: Proceedings of PGAS14, pp 6:1–6:11. ACM
18. Kale LV, Krishnan S (1993) Charm++: a portable concurrent object oriented system based on C++. In: Proceedings of OOSPLA93, pp 91–108. ACM. https://doi.org/10.1145/165854.165874
19. Kasim H et al (2008) Survey on parallel programming model. In: Proceedings of NPC08, pp 266–275. Springer
20. Meyer JC et al Implementation of an energy-aware OmpSs task scheduling policy. http://www.prace-ri.eu/IMG/pdf/wp88.pdf. Accessed 2 May 2017
21. Paudel J et al (2013) On the merits of distributed work-stealing on selective locality-aware tasks. In: Proceedings of ICPP13, pp 100–109. IEEE. https://doi.org/10.1109/ICPP.2013.19
22. Planas J et al (2013) Self-adaptive OMPSS tasks in heterogeneous environments. In: Proceedings of IPDPS13, pp 138–149. IEEE
23. Robison AD (2013) Composable parallel patterns with intel cilk plus. Comput Sci Eng 15(2):66–71. https://doi.org/10.1109/MCSE.2013.21
24. Seo S et al (2017) Argobots: a lightweight low-level threading and tasking framework. IEEE Trans Parallel Distrib Syst 1:1
25. Simmendinger C et al (2015) The GASPI API: a failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures. In: Sustained Simulation Performance, pp 17–32. Springer
26. Subasi O et al (2015) Nanocheckpoints: a task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In: Proceedings of PDP15, pp 99–102. IEEE
27. Wheeler KB et al (2008) Qthreads: an API for programming with millions of lightweight threads. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp 1–8. https://doi.org/10.1109/IPDPS.2008.4536359
28. Willhalm T, Popovici N (2008) Putting intel threading building blocks to work. In: Proceedings of IWMSE08, pp 3–4. ACM. https://doi.org/10.1145/1370082.1370085